A proof-based approach to detect vulnerabilities in C programs

Amel Mammar, Pengfei Liu

▶ **To cite this version:**

Amel Mammar, Pengfei Liu. A proof-based approach to detect vulnerabilities in C programs. SERP 2011 : International Conference on Software Engineering Research and Practice, Jul 2011, Las Vegas, United States. pp.464 - 470. hal-01302477

HAL Id: hal-01302477
https://hal.science/hal-01302477

Submitted on 24 Feb 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Proof-Based Approach to Detect Vulnerabilities in C programs

Amel Mammar[1], **Liu Penfei**[2]
[1]Institut Telecom SudParis, CNRS/SAMOVAR, Paris, France
amel.mammar@it-sudparis.eu
[2]INRIA, Phoenix / Bordeaux-Sud, Talence, France
Pengfei.Liu@inria.fr

Abstract— *This paper presents a formal approach to detect vulnerabilities in a C program using the B formal method. Vulnerabilities denote faults that may be introduced unintentionally into programs making them behave incorrectly. Such faults (or programing errors) may lead to unpredictable behavior and even worse well-motivated attackers may exploit them later to cause real damage. Basically, the proposed approach consists in translating the vulnerable aspects of a C program into a B specification. On this B specification proof and model checking activities are performed in order to detect the presence or absence of vulnerabilities. Compared to the existing vulnerability detection techniques, a proof-based approach permits to eliminate false alarms and denial of service attacks.*

Keywords: Security; Vulnerability detection; Mapping rules; B formal method; Proofs; Model Checking.

1. Introduction

Software vulnerabilities are programming mistakes or bugs that compilers fail to detect. Buffer and arithmetic overflows are well-known examples of vulnerabilities. With the fast proliferation of complex, open and distributed systems, such software vulnerabilities become a real issue that needs to be fixed since these systems are often used in critical domains like transportation, finance, politics, etc; which makes attackers more and more motivated to cause important damage by exploiting these security breaches [6]. Above all, software defects are expensive, according to a 2002 National Institute of Standards and Technology study, software errors cost the U.S. economy an estimated 59.5 USD billion annually [15].

To detect software vulnerabilities, several techniques have been developed [12], [11]. Roughly speaking, these techniques can be classified into two classes: *static* and *dynamic* analysis. Each one has its own particular strengths, however one technique is not sufficient to deal with all the possible errors. In general, several techniques must be applied to detect a larger range of vulnerabilities. The main drawback of static techniques is the high number of false positives they can produce, whereas dynamic techniques suffer from denial of service since the detection is performed at program run-time. A more detailed description of these existing techniques is given in Section 7.

In this paper, we propose a formal approach, based on the B method, to detect vulnerabilities in a C program. Our approach consists in converting the C program into a B specification on which it is possible to perform correctness proof and model checking activities to detect programming bugs or errors. By converting a C program into a B specification, we generate the necessary and sufficient information about the C vulnerable statements. To deal with a buffer overflow for instance, we generate information about the size of the buffer and also the size of the data it holds at any time.

The rest of the paper is structured as follows. The vulnerabilities we consider in this paper are presented in Section 2. The B method is introduced in Section 3. Then, Section 4 gives an overview of the proposal which is described in detail in Section 5. The actual detection of vulnerabilities using proof and model checking activities is described in Section 6. Section 7 presents the related work and a comparison with our approach. Finally, we conclude and present some potential future work.

2. Software vulnerabilities

In [16], several types of vulnerabilities are described together with their countermeasures and how we deal with them using the B method. These vulnerabilities include buffer overflows and different arithmetic overflows and underflows but also format string vulnerabilities. For the sake of space, this paper only presents buffer overflows and arithmetic vulnerabilities.

Buffer overflows. To store the content of variables and buffers, a program reserves a specific amount of memory space. A buffer overflow occurs when a program attempts to put more data in a buffer than it can hold. A buffer is a sequential section of memory allocated to contain anything from a character string to an array of integers. Writing outside the bounds of a block of allocated memory can corrupt data, crash the program, or cause the execution of malicious code. Buffer overflows are the favorite exploit for hackers.

Arithmetic vulnerabilities. These vulnerabilities occur when a calculation produces a value that is greater in magnitude than a given register or storage location can store or represent. Each integer type in C has a fixed minimum and maximum values that depend on the type's machine representation, whether the type is signed or unsigned, and the type's width (e.g., 16-bits vs. 32-bits). At a high level, integer vulnerabilities arise because the programmer does not take into account the maximum and minimum values. Over one hundred C integer vulnerabilities have been publicly identified to date, some of which have resulted in serious disasters such as rocket malfunction.

3. Overview of the B method

Introduced by J. R. Abrial [1], B is a formal method for safe project development. B specifications are organized into abstract machines. Each machine encapsulates state variables on which operations are expressed. The set of the possible states of the system are described using an invariant. The invariant is a predicate in a simplified version of the ZF-set theory, enriched with many relational operators. Operations are specified in the Generalized Substitution Language (GSL) which is similar to the DijkstraŠs guarded command notation. A substitution is like an assignment statement. It allows us to identify which variables are modified by the operation, while avoiding mentioning which ones are not. The generalization allows the definition of non-deterministic and preconditioned substitutions. Refinement is the process of transforming a specification into a less abstract one. A refinement can operate on an abstract machine or another refinement component. In B, we distinguish behavioral and data refinement. In this paper, we only use behavioral refinement that aims at eliminating non-determinism and coming close to the control structures used in the chosen target programming language. Behavioral refinement includes, for example, weakening of preconditions, the replacement of parallel substitution with a sequence one, etc. To ensure the correctness of a B specification, a set of proof obligations is generated for each B component. At the abstract level, these proofs aim at verifying that the invariant of the system is satisfied after the execution of each operation. Of course, such an invariant is assumed to be satisfied before an operation is executed. For each a given invariant Inv and an operation op whose precondition and substitution are P and S respectively, the following proof obligation is raised: $(Inv \wedge P) \Rightarrow [S]Inv$. Notation "$[S]Inv$" denotes the predicate obtained by applying the substitution S to the variables of Inv. Refinement proofs permit us to check the correctness of each concrete operation with respect to its abstraction. We assume that readers are familiar with B method and more details can be found in [1].

4. Overview of the approach to detect vulnerabilities

Our method to deal with vulnerabilities using the B method includes four phases (See Figure 1):

- In a first step, the C code is transformed into an abstract form. To this end, we have defined a sub-set of the C language that include the declarations of buffers and arithmetic data. More details can be found in [16].
- From the abstract representation of the C code, a B formal specification is automatically generated. The B specification contains information about the vulnerable elements of the code. In case of a buffer for instance, we consider its size and also the size of the data it holds at any time.
- To detect the presence/absence of vulnerabilities, proof obligations are generated using the proof obligations generator (POG) of AtelierB. The prover of AtelierB is then used to discharge them. The failure of the prover to automatically demonstrate a goal (the proof) can be due to two different reasons: either it lacks tactics and needs human help and intervention or the goal is false.
- To be sure that an unproved goal corresponds to a vulnerability, we use the model-checker ProB [22] in order to exhibit a state that satisfies the opposite (complement) of this goal. This means that the goal is really false.

The following section describes the generation of a B specification from a C program in order to detect vulnerabilities.

5. Generation of a B model from a C program

In this section, we describe how it is possible to use the B method to detect vulnerabilities in a C program. Our method is based on extracting information about the vulnerable functions or data that are used in the program.

5.1 The subset of ANSI C

Our approach considers a subset of the ANSI C language satisfying the following assumptions:

- the ANSI-C program has been already preprocessed, e.g., all the $\#define$ directives are expanded (inline expansion).

- the supported C types are primitive integer and array.

- function calls are expanded (inline expansion), the return statement is replaced by an assignment (if the function returns a value).

- all the arithmetic operations contain two operands, that is, they are of the form ($z := x\ op\ y$). Multi-operand operations are transformed into binary operations by introducing temporary variables. For instance, operation $(a = a_1 + a_2 + a_3)$ is replaced by the following two binary operations: $a_{temp} = a_1 + a_2$, $a = a_{temp} + a_2$. This

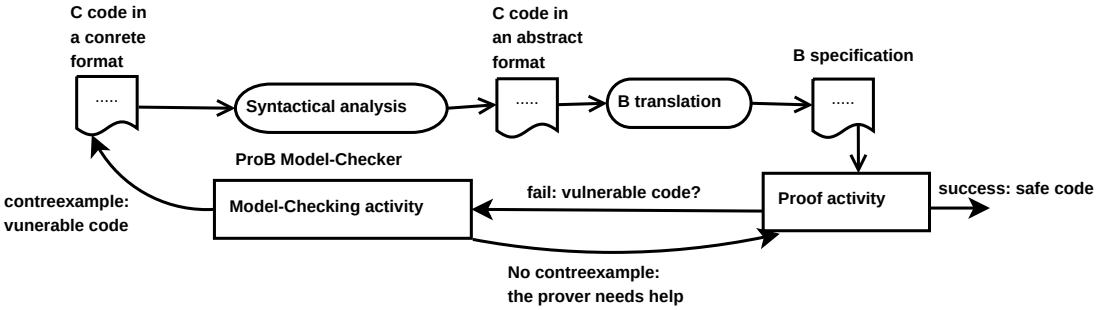

Fig. 1
DETECTING C VULNERABILITIES USING THE B METHOD

transformation is necessary in order to detect any arithmetic overflow. Indeed, multi-operand operations may mask some overflows like in statement $(x = maxint + 1 - 1)$ where the sub-assignment $(x = maxint + 1$ produces an overflow while its equivalent $(x = maxint)$ does not.

- the declaration of variables, buffers and files,

- the predefined C functions like input/output functions ($printf$, $scanf$, etc.) and also the functions on buffers ($strcpy$, $strncpy$) and files ($fscanf$, $fgets$),

- the assignment, the sequence and the choice (**IF**) statements.

A complete grammar of the subset of ANSI C supported by the proposed approach to detect vulnerabilities is provided in [16]. Hereafter, we describe how we obtain the B model from a C program written according to this subset.

5.2 The B model architecture

Recall that our objective is not to build an equivalent B model for a C program. Indeed, our goal is to build an abstraction of a C program by extracting the useful information to detect vulnerabilities that may be present in a C program. Figure 2 depicts the architecture of the B model that we derive from a C program. The B model we derive from a C program is composed of two B components:

- At the abstract level, a B machine is defined. It contains the variables of the C program and an invariant to state the types of these variables but also to express that the C program we would like to build should not contain any vulnerability, that is, all the variables are in their respective ranges. An operation $Main$ is specified to model the behavior part of the C program. This operation consists of a non-deterministic substitution that states that some values are assigned to variables according to the invariant. In other words, the operation represents the behavior of a C program that assigns values to its variables without producing any vulnerability.

- At the refinement level, a B component that refines the first one is created. The refined operation $Main$ of this component contains the translation of the instructions of the C program we are dealing with. By this way, the refinement proofs will demonstrate that the refinement component is conform to its abstract specification, that is, it does not contain any vulnerability.

Buffer declaration	$size\_buff$
char buff[N]	N
char buff[N]="*hello_world*"	N
char buff[]	1
char buff[] = {'*a*','*b*','*c*'}	3
*char *buff ="hello_world"*	11

Table 1
DEFINITION OF THE SIZE OF A BUFFER

In the following sections, we present some translation rules to derive a B model from a C program.

5.3 Translation of buffers

To detect buffer overflows, we have to generate information about the maximum size of each declared buffer and also the amount of the data that it contains at any time. This information is stored in two integer B variables *buffer_max* and *buffer_used* defined as follows:

Constants	*buffer_max*
Properties	*buffer_max=size_buff*
Variables	*buffer_used*
Invariant	*buffer_used* $\in$ NAT $\wedge$ *buffer_used* $\leq$ *buffer_max*

where $size\_buff$ is the size of the declared buffer that is defined according to a set of rules described in [16]. Hereafter, Table 1 gives some examples on how $size\_buff$ is obtained.

Variable $buffer\_used$, initialized to 0, is updated each time a new variable is assigned to the buffer. In [16], we have defined a set of rules that generate a B substitution for each C instruction that modifies the value of the buffer. Table 2 gives some examples.

Substitution (**CHOICE** $buffer\_used := 1$ **OR** $buffer\_used := buffer\_max$ **END**) means that we have to deal with limit cases, that is, the buffer may contain data

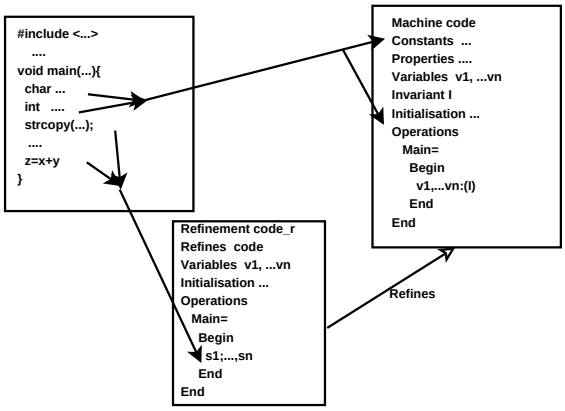

Fig. 2
ARCHITECTURE OF THE B SPECIFICATION

Buffer assignment	**B substitution on** $buffer\_used$
char buff[N]="*hello_world*"	$buffer\_used := 11;$
$gets(buff)$	**CHOICE** $buffer\_used := 1$ **OR** $buffer\_used := buffer\_max$ **END**
$fscanf(file\_name,"\%s",\&buff)$	
$scanf("\%s",\&buff)$	
$fgets(buff, size, fp)$	$buff\_used := size$
$read(fd, buff, size)$	
$strcpy(dest\_buff, source\_buff)$	$dest\_buff\_used := source\_buff\_used$
$strncpy(dest\_buff, src\_buff, size)$	$dest\_buff\_used := size$

Table 2
B TRANSLATION OF C INSTRUCTIONS ON BUFFERS

of minimum or maximum size, this size may be especially equal $buffer\_max$ to simulate a potential attacker. To detect a potential vulnerability at each point of the program, we have to add the following B substitution after each assignment on variable $buffer\_used$

ASSERT ($buffer\_used \leq buffer\_max$) **THEN**
$skip$
END

Type declaration	x_{min}	x_{max}
signed short int x	-32767	32767
unsigned short int x	0	65535
Signed int x	-MAXINT	MAXINT
Unsigned int x	0	2*MAXINT+1

Table 3
MINIMUM AND MAXIMUM VALUES OF SOME C ARITHMETIC TYPES

5.4 Translation of arithmetic data

To detect arithmetic vulnerabilities, we have to verify whether the value assigned to a given variable goes beyond the scope of its type. To do this, we have to memorize the minimum x_{max} and maximum x_{min} values that each variable x can represent. Table 3 gives some examples on how these variables are defined according to the different types of the C language:
For each declared variable x, we generate the following invariant to state that its value must belong to the range of its type: ($x \in x_{min} \,..\, x_{max}$). To detect vulnerabilities, we have also to translate each C instruction that modifies variable x. To get a value from a user, function $scanf()$ is the most common one. Since we have to consider the worst case, each C instruction $scanf("format", \&x)$ is translated into the following B substitutions: **CHOICE** $x := x_{min}$ **OR** $x := x_{max}$ **END**. Other arithmetic operations, like additions, subtractions, multiplications and assignments are straightforwardly translated into B since they are all supported by similar operators with equivalent semantics. As for buffers, after each assignment of variable x, we have to add the following B substitution in order to detect any vulnerability: **ASSERT** ($x \in x_{min} \,..\, x_{max}$) **THEN** $skip$ **END**.

5.5 Translation of C control structures into B

So far, we have seen how to translate declarations (buffer and arithmetic data) and basic C instructions into B. However, a C code may include control structures like conditional (**IF**) and sequential structures. The C control structures we consider in this paper are described by the following BNF grammar:

Instruction ::=
$/* assignment */$
  | $varID = expression$
$/* Sequencing */$
  | $Instruction_1 ; Instruction_2$
$/* choice */$
  | **if** (*expression*) { $Instruction_1$ } **else** { $Instruction_2$ }

Since the B method supports all the above C control structures with the same semantics, the choice, assignment and sequencing instructions are mapped straightforwardly into the **IF**, assignment and sequencing substitutions respectively of the B language.

5.6 Illustrative example

To illustrate our approach, let us apply the different translation rules we have defined on the following C program:

```
#include <stdio.h>
void main(){
    FILE *fp;
    char tmp[40], buff[20];
    fp = fopen("example.txt", "r");
    fgets(tmp,30,fp); strcpy(buff,tmp);
    puts(buff); short int x,y;
    scanf("%h",&x); y=x+2; y=x;}
```

which is translated into:

MACHINE $Example$
CONSTANTS $tmp_{max}, buff_{max}$
PROPERTIES $tmp_{max} = 40 \wedge buff_{max} = 20$
VARIABLES tmp_{used}, $buff_{used}$, x, y
INVARIANT
$tmp_{used} \in NAT \wedge tmp_{used} \leq tmp_{max} \wedge$
$buff_{used} \in NAT \wedge buff_{used} \leq buf_{max} \wedge$
$x \in 0..32767 \wedge y \in 0..32767$
INITIALISATION tmp_{used}, $buff_{used}$, x, $y := 0, 0, 0, 0$
OPERATIONS
Main = **BEGIN**
$tmp_{used}, buff_{used}, x, y :$
$(tmp_{used} \in NAT \wedge tmp_{used} \leq tmp_{max} \wedge$
$buff_{used} \in NAT \wedge buff_{used} \leq buf_{max} \wedge$
$x \in 0..32767 \wedge y \in 0..32767)$
END
END

and

REFINEMENT $Example\_Ref$
REFINES $Example$
VARIABLES tmp_{used}, $buff_{used}$, x, y
INITIALISATION tmp_{used}, $buff_{used}$, x, y:=$0, 0, 0, 0$
OPERATIONS
Main = **BEGIN**
/*translation of fgets(tmp,30,fp)*/
$tmp_{used} := 30$;
ASSERT ($tmp_{used} \leq tmp_{max}$) **THEN** $skip$ **END**;
/*translation of strcpy(buff,tmp)*/
$buff_{used} := tmp_{used}$;
ASSERT ($buff_{used} \leq buff_{max}$) **THEN** $skip$ **END**;
/*translation of scanf("%h",&x)*/
CHOICE $x := 0$ **OR** $x := 32767$ **END**;
$y := x + 2$;
ASSERT ($y \in 0..32767$) **THEN** $skip$ **END**;
$y := x$;
ASSERT ($y \in 0..32767$) **THEN** $skip$ **END**
END
END

6. Vulnerability detection

In this section, we discuss how it is possible to detect vulnerabilities thanks to refinement proofs but also using the ProB tool. To demonstrate the absence of vulnerabilities, we have to discharge all the specification proofs of the abstract component but also those of the refinement component:

1. *Specification proofs*: these proofs aim at demonstrating that operation $Main$, defined in the first abstract level, is correct. We have to prove that this operation re-establishes the invariant: $Inv \Rightarrow [V_1, \ldots V_n : (Inv)]Inv$ where V_i and Inv denote respectively the variables and the invariant of the abstract component. This proof is obvious and automatically discharged.

2. *Refinement proofs*: these proofs aim at demonstrating that the refinement of operation $Main$ does not contradict or violate its abstraction. In other words, we have to prove that the refined operation also re-establishes the invariant: $Inv \Rightarrow [S]Inv$ where S denotes the B translation of the instructions defined in the C program. When establishing these proofs, two cases are possible:

- the proof is discharged (automatically or interactively): this case means that the program is correct and does not contain any vulnerability.

- the prover of AtelierB fails to discharge the proof: this case means that the prover fails to prove a given goal (Predicate) G. Since the complexity of these proofs is very low, such a failure denotes an invariant violation in general, that is, the occurrence of a vulnerability in the corresponding C program. To be sure that this potential vulnerability is a real one, we use the model checking functionality of the ProB tool as a plugins in order to check that the invariant is really violated by finding, for instance, a state that does not satisfy predicate G or contradicts the global invariant.

Let us illustrate this approach on the previous example. For the abstract component $Example$, the POG of AtelierB generates 3 proof obligations that are all automatically discharged as expected. However for the refinement

component $Example\_Ref$, the POG also produces six proof obligations but it succeeds to perform only five of them. The remaining proof obligation is related to invariant $(buff\_used \leq buff\_max)$: $(30 \leq buff\_max)$ which is obviously false. Let us remark that the prover did not point out that invariant $(yy \in 0..32767)$ is also violated. The reason is that the semantics of the **ASSERT** substitution considers its predicate as true to continuous the remaining substitutions. As predicate $(30 \leq buff\_max)$ is not fulfilled, any invariant becomes true ($false \Rightarrow Inv$ is always true). This feature is very interesting in our case since it permits an incremental vulnerability detection. Now, let us replace substitution $(tmp_{used} := 30)$ by $(tmp_{used} := 10)$. The POG generates the following unproved proof related to invariant $(yy \in 0..32767)$: $(xx + 2 \in 0..32767)$. To be sure that this formula is not valid. We use PROB model checker in order to find a state that satisfies its complement $(xx + 2 \notin 0..32767)$. Figure 3 depicts the answer of PROB model checker for the query of finding a state that satisfies the last predicate. Since, a state that violates our goal $(xx+2 \in 0..32767)$ is found, we can conclude that the instruction $(y := x + 2)$ is vulnerable.

7. Related work

To deal with vulnerabilities, several techniques have been developed. We present hereafter those related to buffers and arithmetic data:

- *Buffer overflows vulnerabilities*: the techniques to detect buffer overflows can fall into two categories: *static* and *dynamic*. According to the dynamic technique, the buffer overflow is avoided by restricting the access to the memory of an application. This method primarily relies on a safe code being preloaded before an application is executed. This preloaded component can either provide safer versions of the standard unsafe functions, or it can ensure that the return addresses of the functions are not overwritten. *StackGuard* [5] is an example of the tools following such a method. It places a canary word next to the return address whenever a function is called. If the canary word has been altered when the function returns, we are sure some attempt has been made on the overflow buffers. In that case, it responds by emitting an alert message and halting the program. This technique is also used by *StackShield*[1], *SFI* [20], *libsafe*[2] and *GCC*. Another approach developed by Arash Baratloo et. al. [2] consists in replacing unsafe library functions with safe implementations. However, run-time solutions always incur some performance penalty(StackGuard reports performance overhead up to 40% [7]). The other problem with run-time solutions is that while they may be able to detect or prevent a buffer overflow attack, they effectively turn it into a denial-of-service attack.

[1]http://www.angelfire.com/sk/stackshield

[2]http://www.research.avayalabs.com/project/libsafe

Static techniques permit to overcome this problem by detecting likely vulnerabilities before deployment. RATS [9] and ITS4 [19] are examples of tools based on the static analysis technique. They use lexical analysis and compare the identified lexemes with a "suspects" database to identify vulnerabilities in C source files. The main drawback of such tools is that they may generate false warnings and sometimes miss real problems. Another tool, CodeAuditor[21], uses Model checking to find the vulnerabilities. However as we know, if the code is very large, we can not finish the verification in an acceptable period of time. If we reduce the code source size, we may miss some vulnerabilities also.

- *Integer vulnerabilities*: one approach to fix integer vulnerabilities is to raise a compile-time warning for each potential vulnerability and let the programmer fix them. However, the number of actual vulnerabilities is an order of magnitude less than the number of warnings. Another approach consists in translating the C code into a type-safe code, e.g., *Cyclone* [10] or *CCured* [14]. However, this option may not be practical in many settings, such as for performance-critical applications or when the user is not familiar with the type-safe code. *PICK* [3] is a tool that uses sub-typing rules to detect unsafe integer operations and inserts the necessary dynamic checks to prevent exploits. But it only deals with the casting vulnerabilities. *RICH* [4] uses a similar approach to *PICK* and also suffers of denial of service attacks.

In this paper, we define a formal approach to detect buffer and arithmetic vulnerabilities using the B method. Our approach consists of a static analysis of the code in order to extract the sufficient and the necessary information about the different elements that may produce vulnerabilities. This information is then modeled in B in order to prove the presence/absence of vulnerabilities. Based on a static technique and a formal method, our approach has the advantage of avoiding denial of service attacks and false alarms at the same time.

8. Conclusion and future work

Software security has always been a concern in the software industry. The C language is one of the most used program language in the world, also its vulnerabilities are well-known. In this paper, we have defined a formal method based on the B method in order to detect vulnerabilities with the ultimate goal of correcting them. Basically, our proposal consists in defining variables to store the size of buffers, and also to store the values each integer variable can represent. These different informations together with the C program are then translated into a B specification on which we perform proof and model checking activities to detect the presence of vulnerabilities. We have applied this approach on several applications (larger and more complex than the example given in this paper) that gave very promising results.

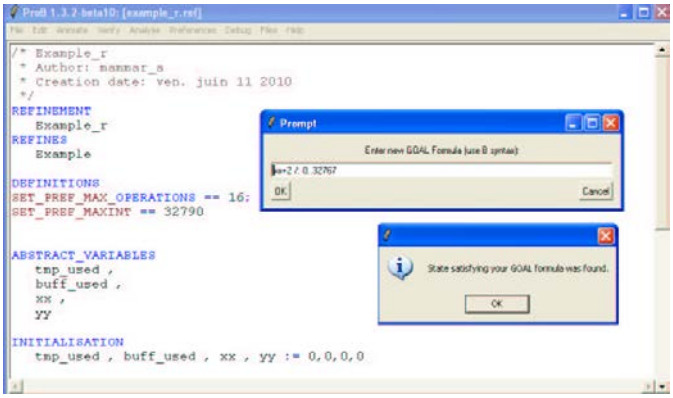

Fig. 3

DETECTION OF VULNERABILITIES USING PROB MODEL CHECKER

Compared to other approaches, our approach can detect buffer overflow and Integer vulnerabilities at the same time without generating false alarms. In addition, our proposal can be applied on any other programming language, like JAVA for instance, by defining new translation rules to deal with its vulnerable functions. Finally to make our approach workable, we have developed a tool that permits us to point out the vulnerable statements in a C program. This tool is implemented in JAVA and described elsewhere in [16]. Currently, we are working on extending the proposed approach to support loops and recursive functions. Future work include also the development of a process that would permit to correct automatically the C code by exploring the B proof and model-checking result activities. Basically, we have to define the condition to add within the $Main$ operation in order to establish the invariant and correct the corresponding C program. To do that, we can reuse our previous work presented in [13] that defines a formal process to calculate the weakest precondition of a B operation to re-establish an invariant.

References

[1] ABRIAL J. R.: The B-Book: Assigning Programs to Meanings, Cambridge University Press, (1996).

[2] Baratloo, A., Singh, N., Tsai, T.: Transparent Run-Time Defense Against Stack-Smashing Attacks Protecting Systems from Stack Smashing Attacks with StackGuard, the 9^{th} USENIX Security Symposium, (2000).

[3] Brumley, D., Song, D., Slember, J.: Towards Automatically Eliminating Integer-Based Vulnerabilities, Technical Report, School of Computer Science, Carnegie Mellon University, (2006).

[4] Brumley, D., Xiaodong, D., Song and Tzi-cker Chiueh and Rob Johnson and Huijia Lin RICH: Automatically Protecting Against Integer-Based Vulnerabilities, the 14^{th} Annual Network & Distributed System Security Symposium(NDSS), (2007).

[5] Calton, C.C., Pu C., Maier D., Hinton H., Walpole, J., Bakke, P., Beattie S., Grier, A., Wagle, P., Zhang, Q.: StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks, the 7^{th} USENIX Security Symposium, (1998).

[6] CERT Coordination Center. CERT/CC statistics http://www.cert.org/stats/historical.html, (2007).

[7] Cowan, C., Beattie, S., Day, R.F., Pu, C., Wagle, P., Walthinsen, E.: Protecting Systems from Stack Smashing Attacks with StackGuard, the 7^{th} USENIX Security Symposium, (1998).

[8] Cowan, C., Barringer, M., Beattie, S., Kroah-Hartman, G.: FormatGuard : Automatic Protection From printf Format String Vulnerabilities, the 10^{th} USENIX Security Symposium, (2001).

[9] Fortify Software Inc Secure software solutions, rats, the rough auditing tool for security, http://www.securesw.com/rats, (2001).

[10] Jim, T., Morrisett, J.G., Grossman, D., Hicks, M.W. Cheney, J., Wang, Y.: Cyclone: A Safe Dialect of C, the 9^{the} USENIX Annual Technical Conference, (2002).

[11] Jimenez, W., Mammar, A., Cavalli, A.R.: Software Vulnerabilities, Prevention and Detection Methods: A Review, the 1^{first} International Workshop on Security in Model Driven Architecture(SEC-MDA), (2009).

[12] Kuang, C., Miao, Q., Chen, H.: Analysis of Software Vulnerability, the 5^{th} WSEAS International Conference on Information Security and Privacy(ISP), (2006).

[13] X, Y.: A Systematic Approach to Generate B Preconditions: Application to the Database Domain, Software and System Modeling 8(3), (2009).

[14] Necula, G., McPeak, S., Weimer, W.: CCured: Type-Safe Retrofitting of Legacy Code, the 29^{th} Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), (2002).

[15] NIST. The Economic Impacts of Inadequate Infrastructure for Software Testing. National Institute of Standards and Technology-Final Report, (2002).

[16] X, Y.: Detecting C Program Vulnerabilities Master Thesis, Telecom SudParis, (2009). Available at http://phoenix.inria.fr/index.php/members/36-pengfei-liu.

[18] Shankar, U., Talwar, K., Foster, J.S., Wagner, D.: Detecting Format String Vulnerabilities with Type Qualifiers, the 10^{th} conference on USENIX Security Symposium(SSYM), (2001).

[19] Viega, J., Bloch, J. T., Kohno, Y., McGraw, G.: ITS4: A Static Vulnerability Scanner for C and C++ Code, the 16^{th} IEEE Annual Conference Computer Security Applications(ACSAC), (2000).

[20] Wahbe, R., Lucco, S., Anderson, T.E., Graham, S.L.: Efficient Software-Based Fault Isolation, the 14^{th} ACM Symposium on Operating Systems Principles(SOSP), (1993).

[21] Wang, L., Zhang, Q., Zhao, P.: Automated Detection of Code Vulnerabilities Based on Program Analysis and Model Checking, the 8^{th} IEEE International Working Conference on Source Code Analysis and Manipulation(SCAM), (2008).

[22] ProB.: http://users.ecs.soton.ac.uk/mal/systems/prob.html.